Documentazione

Pierluigi Zarra, Leonardo Oliva e Vishes Radhakeesoon

TripPlanner

18 Febbraio 2025

Indice

Sommario

Questo elaborato illustra l'implementazione di TripPlanner, un'applicazione web Full-Stack per la gestione di itinerari turistici condivisi. L'obiettivo del progetto è dimostrare l'efficacia di un'architettura Event-Driven applicata a sistemi collaborativi moderni.

Il backend, basato su Node.js ed Express, espone un set di API REST protette e gestisce la persistenza dei dati su database NoSQL (MongoDB). La comunicazione asincrona tra client e server è potenziata dall'uso di Socket.io.

Lato frontend, l'applicazione è costruita con React.js e sfrutta l'ecosistema Vite. Il progetto integra API di terze parti per il geocoding e il routing (OSRM) e implementa pratiche di sicurezza avanzate.

1. Introduzione

1.1. Contesto Applicativo: L'Evoluzione dell'E-Tourism

Il progetto TripPlanner si inserisce nel contesto dell'E-Tourism e delle Social Web Applications. Negli ultimi anni, il settore turistico ha subito una profonda trasformazione guidata dall'evoluzione del Web, passando da un modello statico (Web 1.0), dove l'utente era un semplice fruitore di informazioni, a un modello dinamico e partecipativo (Web 2.0).

In questo scenario, l'utente moderno non è più solo un consumatore, ma un *prosumer* (produttore e consumatore) che desidera pianificare attivamente le proprie esperienze, condividere itinerari e collaborare con altri viaggiatori. Tuttavia, nonostante l'abbondanza di strumenti per la prenotazione (es. Booking, Skyscanner), mancano soluzioni integrate efficaci per la fase di pianificazione collaborativa di gruppo.

Studi recenti nell'ambito del Computer-Supported Cooperative Work (CSCW) evidenziano come la pianificazione collaborativa riduca lo stress decisionale nei gruppi (Autore, Anno) (da finire).

1.2. Definizione del Problema

L'idea alla base di TripPlanner nasce dall'analisi di una problematica comune nell'organizzazione di viaggi di gruppo: la frammentazione dell'informazione. Attualmente, il coordinamento di un itinerario condiviso avviene spesso attraverso l'utilizzo improprio di strumenti non dedicati:

- **Canali di comunicazione asincroni:** Chat di gruppo (WhatsApp, Telegram) dove informazioni cruciali come orari, indirizzi e conferme si perdono nel flusso dei messaggi ("scroll infinito").
- **Strumenti statici:** Note condivise o fogli di calcolo (Excel) che mancano di interattività, non offrono visualizzazione geografica e non supportano aggiornamenti in tempo reale.

Questa dispersione informativa porta spesso a disallineamenti tra i partecipanti, difficoltà nel visualizzare la logistica degli spostamenti e una generale inefficienza organizzativa.

1.3. Obiettivo e Soluzione Proposta

L'obiettivo di TripPlanner è risolvere queste criticità fornendo una piattaforma centralizzata che aggreghi in un'unica Dashboard interattiva tutte le fasi decisionali del viaggio. La soluzione proposta è una Single Page Application (SPA) progettata per offrire un'esperienza utente fluida e reattiva, che permette di:

1. **Centralizzare i dati:** Creare itinerari dettagliati dove ogni tappa è strutturata e modificabile.
2. **Visualizzare il contesto spaziale:** Sfruttare mappe interattive (tramite integrazione Leaflet) per dare immediata evidenza delle distanze e dei percorsi.

3. **Collaborare in Real-Time:** Gestire le adesioni e le modifiche con un sistema di notifiche istantanee (basato su Socket.io), garantendo che tutti i partecipanti siano sempre sincronizzati sullo stato più recente del viaggio.

1.4. Innovazione Tecnologica

Dal punto di vista tecnico, il progetto è stato sviluppato adottando lo stack MERN (MongoDB, Express, React, Node.js), una scelta che risponde ai requisiti moderni di scalabilità e manutenibilità. L'architettura modulare del frontend e l'uso di un database NoSQL documentale permettono di gestire con flessibilità strutture dati complesse (come itinerari multi-tappa e liste partecipanti dinamiche), offrendo al contempo un'interfaccia responsive e accessibile.

2. Obiettivi del Progetto

Il progetto TripPlanner è stato ideato con una duplice finalità: da un lato, soddisfare esigenze operative legate alla gestione collaborativa dei viaggi (**Obiettivi Funzionali**); dall'altro, implementare un'architettura software moderna, scalabile e conforme agli standard di sviluppo web attuali (**Obiettivi Tecnici**).

2.1. Obiettivi Funzionali

L'applicazione è progettata per coprire l'intero ciclo di vita dell'organizzazione di un viaggio, trasformando un processo solitamente frammentato in un flusso di lavoro unificato.

- **Pianificazione e Gestione Itinerari (CRUD):** L'obiettivo primario è fornire un'interfaccia intuitiva per la creazione, lettura, modifica e cancellazione dei viaggi. Il sistema permette di definire parametri essenziali quali origine, destinazione, date, mezzo di trasporto e visibilità (Pubblica/Privata). Una caratteristica distintiva è la gestione dinamica delle tappe intermedie, che consente di strutturare itinerari complessi multi-destinazione.
- **Integrazione Geospaziale e Routing:** Per migliorare la percezione spaziale del viaggio, l'applicazione integra servizi di mappatura interattiva (Leaflet su OpenStreetMap). Oltre alla semplice visualizzazione dei marker, il sistema implementa un algoritmo di routing (tramite API OSRM) che calcola e traccia automaticamente il percorso stradale tra le tappe inserite, fornendo anche una stima della durata del viaggio per le tratte in auto.
- **Gestione Sociale e Ruoli (RBAC Semplificato):** Il sistema implementa una logica di accesso basata sui ruoli per garantire la corretta gestione del gruppo:
  - **Creator (Amministratore):** Ha il controllo totale sul viaggio, inclusa la facoltà di modificare l'itinerario, rimuovere partecipanti indesiderati ("kick") e accettare richieste di adesione.
  - **Partecipante:** Può visualizzare i dettagli completi e abbandonare il gruppo.
  - **Utente Esterno:** Può inviare richieste di partecipazione (per viaggi pubblici) o essere invitato tramite email (per viaggi privati).
- **Interattività Real-Time:** Per superare la latenza tipica delle applicazioni web tradizionali, TripPlanner integra un sistema di notifiche istantanee basato su protocollo WebSocket (Socket.io). Questo permette agli utenti di ricevere feedback immediati (es. "L'utente X vuole unirsi al viaggio") senza la necessità di ricaricare la pagina, migliorando drasticamente l'esperienza utente (UX).

2.2. Obiettivi Tecnici e Architetturali

Dal punto di vista ingegneristico, il progetto mira a realizzare una Single Page Application (SPA) robusta, sicura e manutenibile, adottando lo stack tecnologico MERN (MongoDB, Express, React, Node.js).

- **Architettura Frontend a Componenti:** Lo sviluppo in React segue rigorosamente il pattern *Separation of Concerns*, dividendo l'applicazione in:
  - **Container Components:** (es. `CreateTrip.jsx`, `Dashboard.jsx`) Responsabili della logica di business, gestione dello stato e interazione con le API.
  - **Presentational Components:** (es. `TripForm.jsx`, `TripCard.jsx`) Componenti puri, privi di logica di business, dedicati esclusivamente al rendering dell'interfaccia utente in base alle props ricevute.
- **Gestione dello Stato e Side Effects:** L'applicazione sfrutta i moderni React Hooks (`useState`, `useEffect`, `useCallback`) per gestire il ciclo di vita dei componenti e le operazioni asincrone. Particolare attenzione è stata posta nell'ottimizzazione delle performance, come l'uso del *debouncing* nella barra di ricerca per limitare le chiamate al server durante la digitazione.
- **Design System Scalabile:** L'interfaccia utente è stata costruita su un sistema di variabili CSS globali (`:root`), che centralizza la gestione di colori, spaziature e tipografia. Questo approccio ha permesso l'implementazione nativa del Dark Mode, garantendo coerenza visiva e facilità di manutenzione del codice di stile.
- **Sicurezza e Autenticazione Stateless:** La sicurezza è garantita tramite un sistema di autenticazione basato su JSON Web Token (JWT).
  - **Hashing:** Le password degli utenti vengono crittografate tramite bcrypt prima di essere salvate nel database.
  - **Middleware:** Le rotte sensibili del backend sono protette da middleware (`verifyAccessToken`) che intercettano e validano il token presente nell'header delle richieste HTTP.
  - **Verifica Identità:** È stato implementato un flusso di verifica account tramite token inviato via email (Nodemailer), per garantire la legittimità delle registrazioni.

3. Modello di Processo e Valutazioni Iniziali

3.1. Modello di Processo Adottato

Per lo sviluppo di TripPlanner è stato adottato un modello di ciclo di vita del software di tipo **Iterativo e Incrementale**, fortemente ispirato alle metodologie Agile[cite: 56]. Questa scelta è stata dettata dalla natura del progetto: essendo un lavoro sviluppato da zero con un team in fase di apprendimento, non era possibile definire rigidamente tutti i requisiti all'inizio.

Il lavoro è stato suddiviso in quattro iterazioni funzionali principali ("Sprint"):

1. **Iterazione Core:** Setup dell'ambiente, configurazione del database e sistema di autenticazione JWT.
2. **Iterazione Dati:** Implementazione delle operazioni CRUD per la gestione dei viaggi.
3. **Iterazione Visuale:** Integrazione delle mappe interattive (Leaflet) e logica di routing.
4. **Iterazione Real-Time:** Sviluppo del sistema di notifiche e gestione della comunicazione tramite socket.

3.2. Pianificazione Temporale (Gantt)

La pianificazione delle attività ha coperto un arco temporale di circa 12 settimane. La distribuzione delle fasi di lavoro è illustrata nel diagramma seguente:

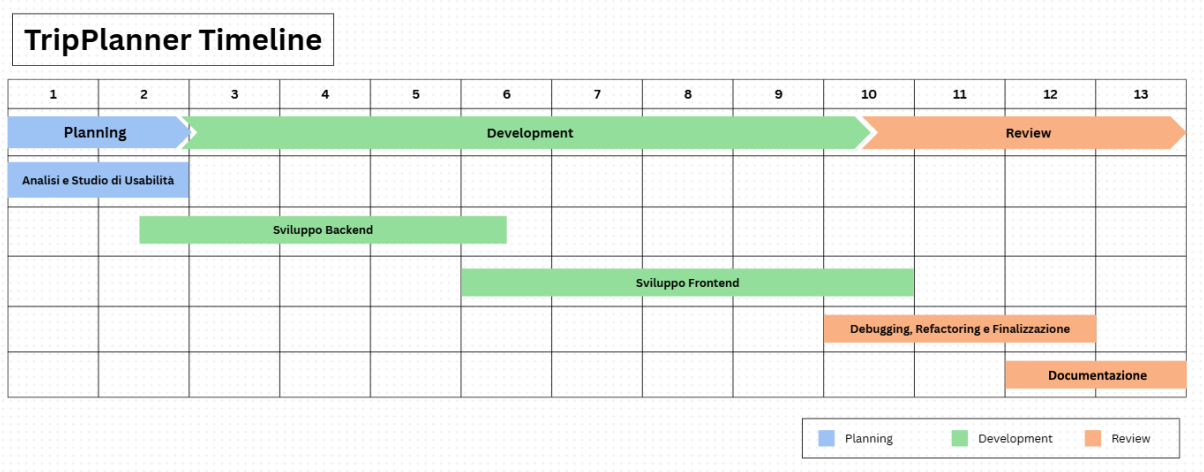

Figura 1: Diagramma di Gantt delle fasi di sviluppo di TripPlanner.

- **Settimane 1-2:** Analisi dei requisiti, studio UI/UX (inclusa la scelta della Dark Mode) e analisi delle API cartografiche.
- **Settimane 2-7:** Sviluppo Backend, inclusa la modellazione degli schemi MongoDB (User, Itinerario, Notifica) e le API RESTful.

- **Settimane 6-10:** Sviluppo Frontend con React e integrazione della mappa interattiva.
- **Settimane 10-12:** Testing, implementazione finale dei WebSocket per le notifiche e pulizia del codice.
- **Settimane 12-13:** Stesura della documentazione.

3.3. Analisi dei Rischi e Strategie di Mitigazione

Durante lo sviluppo sono stati identificati e gestiti diversi fattori di rischio:

- **Rischi di Competenza:** La necessità di apprendere concetti complessi (es. autenticazione Stateless) durante l'implementazione è stata mitigata tramite *Code Review* periodiche tra i membri del gruppo.
- **Rischi Esterni:** Per gestire possibili disservizi delle API di terze parti (OpenStreetMap/OSRM), sono stati implementati blocchi `try-catch` nel frontend per prevenire crash e avvisare l'utente.
- **Rischi Organizzativi:** Le difficoltà di coordinamento sincrono sono state superate grazie all'architettura disaccoppiata (Backend API vs Frontend SPA), che ha permesso uno sviluppo asincrono e parallelo dei moduli.
- **Rischi Tecnologici:** Per tecnologie nuove come Socket.io, il team ha utilizzato piccoli prototipi separati (*Proof of Concept*) prima dell'integrazione nel progetto principale.

4. Metodo Proposto e Requisiti

4.1. Metodologia di Analisi

La definizione delle specifiche del sistema TripPlanner è avvenuta attraverso un processo di analisi incrementale. Partendo dai requisiti macroscopici definiti nella traccia d'esame, sono stati dettagliati i casi d'uso specifici per garantire un'esperienza utente completa.

Il sistema è stato modellato attorno a tre attori principali:

1. **Utente Visitatore:** Può visualizzare l'home dell'app e i viaggi disponibili ma non può interagire con i dati.
2. **Utente Registrato:** Può creare viaggi, gestire il proprio profilo e richiedere di partecipare a viaggi altrui.
3. **Organizzatore (Creator):** L'utente che ha creato lo specifico viaggio; possiede privilegi amministrativi per la modifica, cancellazione e gestione dei partecipanti.

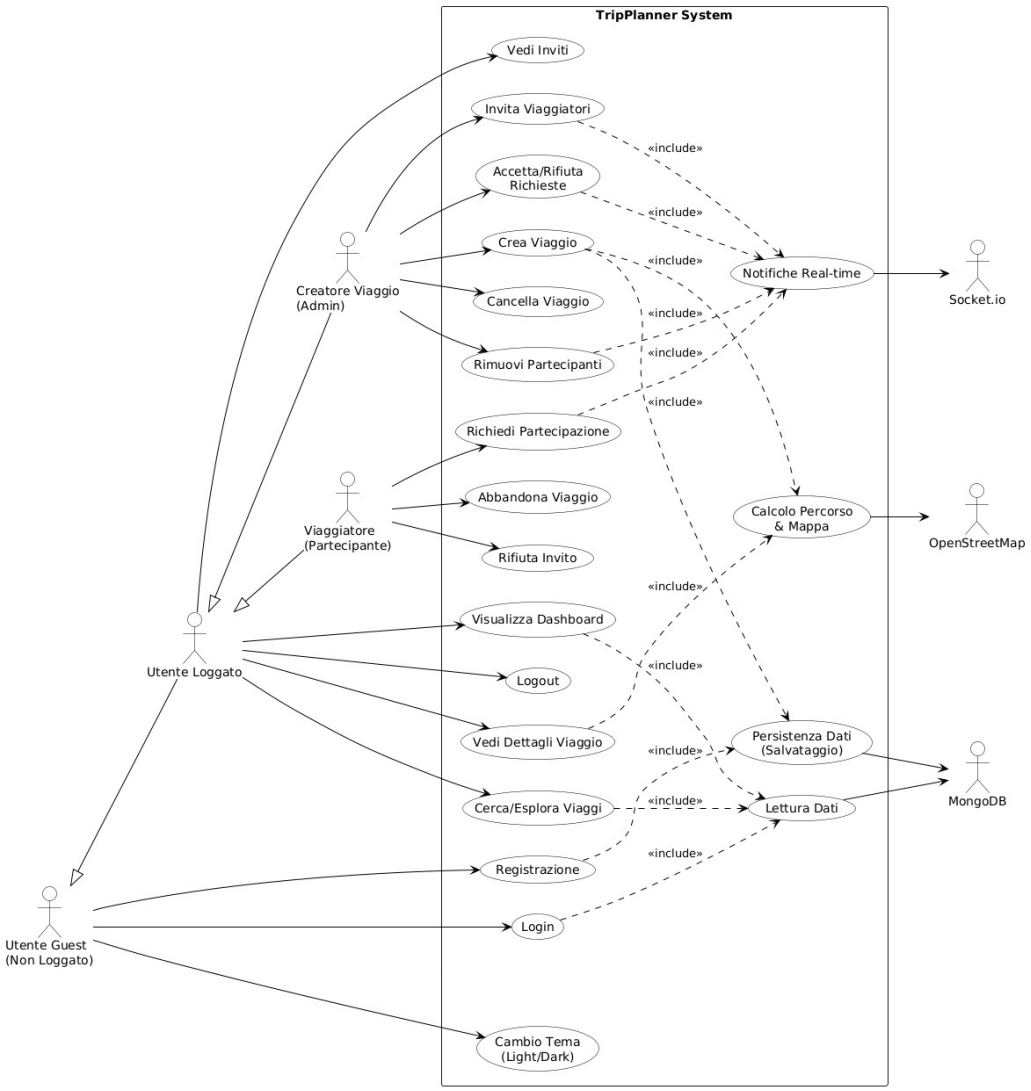

Figura 2: Diagramma dei casi d'uso del sistema TripPlanner.

4.2. Requisiti Funzionali (FR)

I requisiti funzionali descrivono le interazioni specifiche che il sistema deve supportare:

- **FR1 - Gestione Identità e Sicurezza:**
  - Registrazione e Login tramite email/password o Google OAuth 2.0.
  - Verifica account tramite token inviato via email.
  - Gestione sessione sicura tramite Access e Refresh Token.
- **FR2 - Gestione Viaggi:**
  - Operazioni CRUD complete per gli itinerari.
  - Dettagli strutturati: date, mezzo di trasporto, visibilità e tappe intermedie[cite: 197].
  - Ricerca con *debouncing* per ottimizzare le performance del server.
- **FR3 - Integrazione Geospaziale:**
  - Visualizzazione interattiva dei marker sulla mappa (Leaflet).
  - Calcolo automatico della rotta stradale tramite API OSRM per i viaggi in auto.
- **FR4 - Collaborazione:**
  - Sistema di partecipazione differenziato per viaggi pubblici (richiesta) e privati (invito).
  - Gestione del gruppo con facoltà di rimozione ("kick") o abbandono volontario.
  - Notifiche istantanee via WebSocket per eventi critici[cite: 210].

4.3. Requisiti Non Funzionali (NFR)

I requisiti non funzionali definiscono gli attributi di qualità del sistema[cite: 212]:

- **NFR1 - Usabilità:** Interfaccia Responsive e supporto nativo alla Dark Mode[cite: 213, 214].
- **NFR2 - Sicurezza:** Hashing delle password con bcrypt e protezione delle API tramite middleware JWT.
- **NFR3 - Performance:** Navigazione fluida tipica delle SPA senza ricaricamenti di pagina.
- **NFR4 - Robustezza:** Gestione elegante degli errori lato client per evitare crash dell'interfaccia.

5. Architettura del Sistema e Tecnologie Utilizzate

Questo capitolo illustra le scelte progettuali effettuate per la realizzazione di TripPlanner, descrivendo l'architettura logica del sistema e lo stack tecnologico selezionato per l'implementazione.

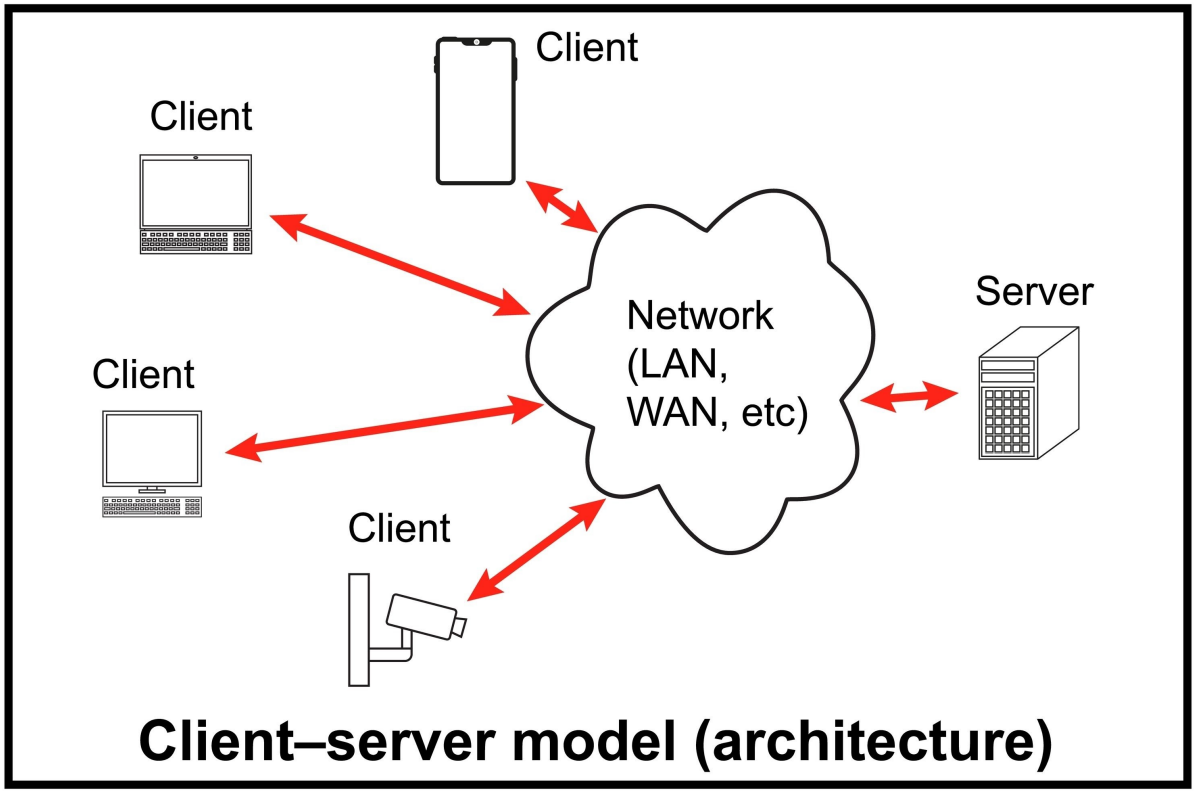

Figura 3: Esempio di Architettura a Nanoservizi

5.1. Architettura del Sistema

Il sistema TripPlanner è stato progettato seguendo un'architettura Client-Server moderna basata sul modello REST (*Representational State Transfer*) per la gestione delle risorse e sul protocollo WebSocket per la comunicazione in tempo reale.

L'applicazione è strutturata come una **Single Page Application (SPA)**. A differenza delle applicazioni web tradizionali, dove ogni interazione richiede il ricaricamento completo della pagina dal server, la SPA carica una singola pagina HTML iniziale e aggiorna dinamicamente il contenuto in risposta alle azioni dell'utente. Questo garantisce un'esperienza utente (UX) fluida e reattiva, simile a quella di un'applicazione desktop nativa.

5.1.1. Diagramma Architetturale

Il sistema è organizzato secondo il pattern *3-Tier Architecture* (Architettura a tre livelli), come illustrato nel diagramma seguente:

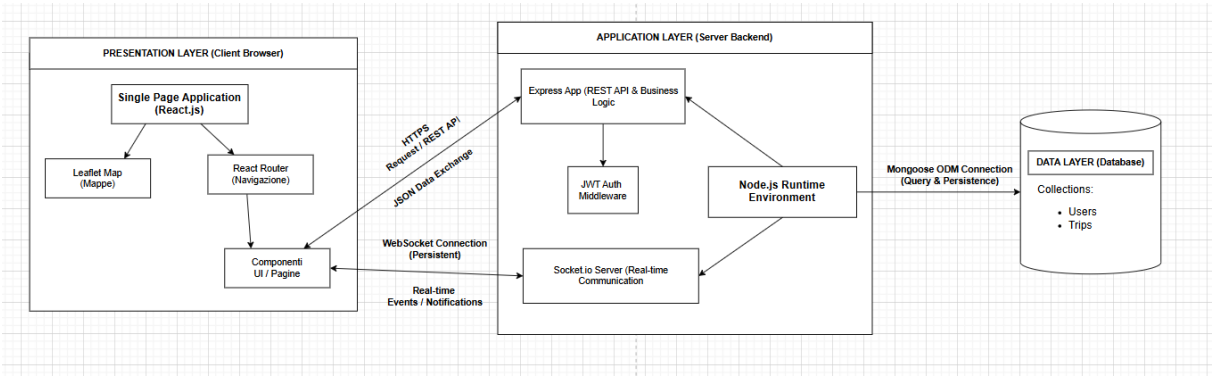

Figura 4: Diagramma dell'architettura logica di TripPlanner e flusso dei dati.

Come evidenziato nel diagramma, il sistema si divide in:

1. **Presentation Layer (Client):** È il punto di interazione con l'utente. Esegue il codice React nel browser, gestisce la visualizzazione delle mappe e invia richieste al server.

2. **Application Layer (Server):** Il server Node.js/Express funge da intermediario. Riceve le richieste HTTP o gli eventi Socket.io, esegue la logica di business e comunica con il database.

3. **Data Layer (Database):** MongoDB archivia i dati in modo persistente.

I canali di comunicazione utilizzati sono due:

- **Canale HTTP (Linea continua):** Utilizzato per operazioni standard (Login, creazione viaggio, salvataggio tappe).

- **Canale WebSocket (Linea tratteggiata):** Un canale bidirezionale persistente utilizzato per le notifiche push in tempo reale tra server e client.

5.2. Stack Tecnologico

Per lo sviluppo di TripPlanner è stato adottato lo stack **MERN** (MongoDB, Express, React, Node.js). Questa combinazione "Full Stack JavaScript" offre il vantaggio dell'uniformità linguistica, permettendo di utilizzare JavaScript (ES6+) sia per lo sviluppo frontend che backend, oltre all'utilizzo del formato JSON per lo scambio dati.

5.2.1. Frontend (Interfaccia Utente)

Il lato client è responsabile della presentazione e dell'interattività:

- **React.js:** Libreria JavaScript dichiarativa basata su componenti. Scelta per la gestione di interfacce complesse e per il Virtual DOM che ottimizza le performance.

- **React Router:** Gestisce il routing lato client, permettendo la navigazione tra le viste (Dashboard, Mappa, Profilo) mantenendo sincronizzato l'URL senza ricaricare la pagina.

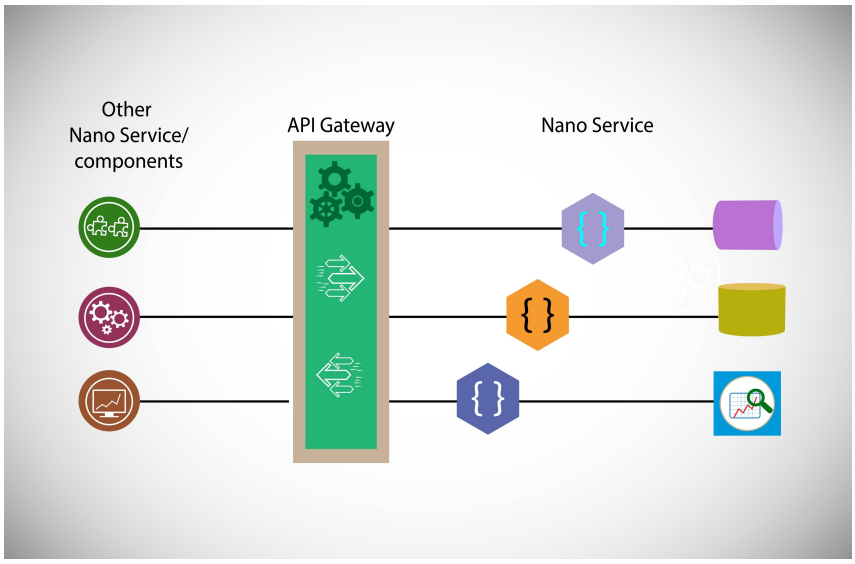

Figura 5: Diagramma Client-Server.

- **Leaflet & React-Leaflet:** Libreria open-source per le mappe interattive. Offre un'alternativa leggera a Google Maps, integrandosi con OpenStreetMap.
- **Axios:** Client HTTP basato su Promise per effettuare chiamate asincrone alle API del backend.
- **CSS Modules & Variabili CSS:** Approccio modulare senza framework pesanti, utilizzando variabili CSS native per gestire il tema dinamico (Light/Dark Mode).

5.2.2. Backend (Logica Applicativa)

Il lato server gestisce la logica di business e la sicurezza:

- **Node.js:** Runtime environment ideale per gestire molteplici connessioni simultanee grazie alla sua architettura *event-driven* e *non-blocking I/O*.
- **Express.js:** Framework minimalista per strutturare le rotte API RESTful e gestire i middleware.
- **Socket.io:** Libreria che abilita la comunicazione bidirezionale real-time, fondamentale per il sistema di notifiche istantanee.
- **JWT (JSON Web Token):** Standard per l'autenticazione *stateless*, che verifica l'identità dell'utente ad ogni richiesta senza memorizzare sessioni sul server.

5.2.3. Database (Persistenza dei Dati)

- **MongoDB:** Database NoSQL orientato ai documenti. La struttura JSON-like (BSON) si adatta perfettamente alla natura variabile dei dati di viaggio (tappe, partecipanti dinamici).
- **Mongoose:** ODM (*Object Data Modeling*) per Node.js. Utilizzato per definire schemi rigorosi, gestire validazioni e semplificare le query.

5.3. Modellazione dei Dati (Data Model)

La persistenza dei dati è affidata a MongoDB. Nonostante la natura "schemaless", l'applicazione utilizza Mongoose per definire schemi rigorosi a livello applicativo. Il database è organizzato in tre collezioni principali interconnesse tramite References: `Users`, `Itinerarios` e `Notifications`.

5.3.1. Schema Entità-Relazione (ER)

Il sistema utilizza un approccio relazionale ibrido: i viaggi mantengono riferimenti agli utenti (`ObjectId`) per gestire i partecipanti e le notifiche fungono da collegamento per le interazioni asincrone.

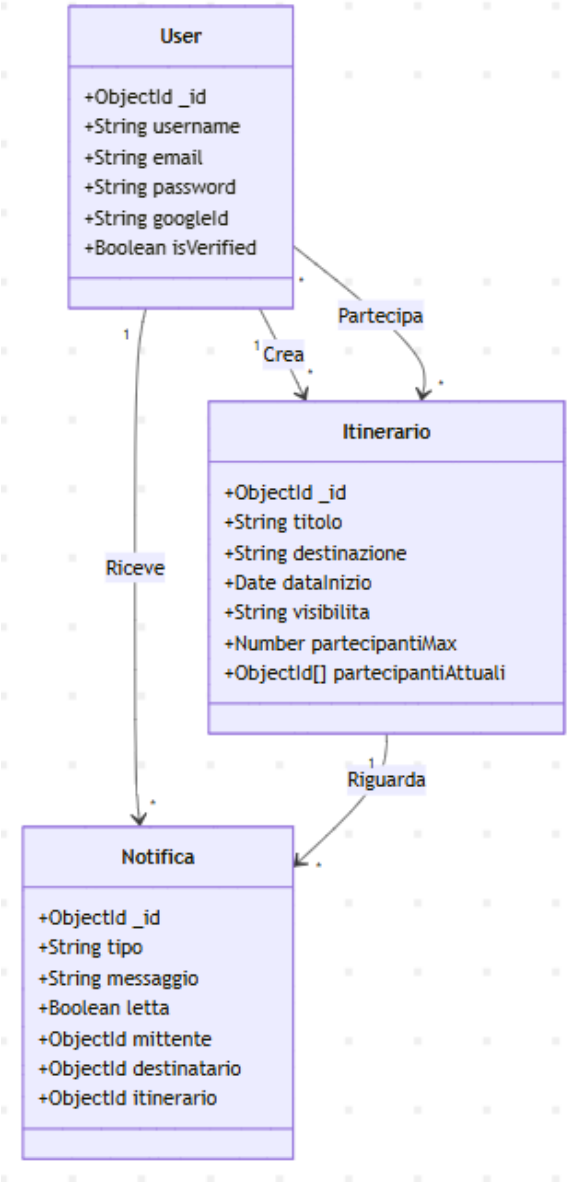

Figura 6: Diagramma Entità-Relazione (ER) del database di TripPlanner.

5.3.2. Dettaglio delle Collezioni

Tabella 6.1: Collezione Users

Rappresenta l'identità dell'utente e supporta l'autenticazione ibrida (Locale + Google OAuth).

Campo	**Tipo**	**Descrizione**	**Note e Vincoli**
username	String	Nome visualizzato	Required, Trimmed
email	String	Indirizzo email univoco	Unique, Required
password	String	Hash della password (bcrypt)	Required (se non Google), Min 6 char
googleID	String	ID per login Google	Unique, Sparse
isVerified	Boolean	Stato verifica email	Default: false
verificationToken	String	Token temporaneo per link email	-

Tabella 6.2: Collezione Itinerarios

È l'entità centrale del sistema. Gestisce sia i dettagli del viaggio che la logica sociale.

Campo	**Tipo**	**Descrizione**	**Note e Vincoli**
titolo	String	Nome del viaggio	Max 100 char
autore	ObjectId	Creatore del viaggio	Ref: User
destinazione	String	Luogo principale	Required
visibilita	String	Privacy del viaggio	Enum: 'pubblica', 'privata'
travelMode	String	Mezzo di trasporto	Default: 'FLIGHT'
partecipantiMax	Number	Limite posti disponibili	Min: 1
partecipantiAttuali	[ObjectId]	Lista utenti confermati	Ref: User
richiestePendenti	[ObjectId]	Utenti in attesa	Ref: User
condivisoCon	[String]	Email per inviti privati	Array di stringhe

Tabella 6.3: Collezione Notifications
Gestisce le interazioni asincrone tra gli utenti.

Campo	**Tipo**	**Descrizione**	**Note e Vincoli**
destinatario	ObjectId	Chi riceve la notifica	Ref: User (Required)
mittente	ObjectId	Chi ha generato l'evento	Ref: User
itinerario	ObjectId	Viaggio oggetto della notifica	Ref: Itinerario
tipo	String	Categoria dell'evento	Enum: 'richiesta...', 'invito...'
messaggio	String	Testo descrittivo	Opzionale
letta	Boolean	Stato lettura	Default: false

Infine, esiste una collezione tecnica `refreshTokens` utilizzata per aumentare la sicurezza dell'autenticazione JWT, permettendo di revocare l'accesso (Logout) invalidando il token nel database.

6. Prototipazione e Setup Iniziale

In questo capitolo viene descritta la fase operativa dello sviluppo, partendo dalla configurazione dell'ambiente di lavoro (*Scaffolding*) fino all'evoluzione dell'interfaccia grafica. Questo processo iterativo ha permesso di trasformare i requisiti architetturali definiti nel capitolo precedente in una struttura di codice concreta e in un'esperienza utente rifinita.

6.1. Organizzazione del Codice (Project Scaffolding)

La prima fase di sviluppo ha riguardato l'impostazione della struttura delle directory. Per mantenere una netta separazione delle responsabilità, il progetto è stato organizzato in due directory principali distinte: `backend` e `frontend`, gestite all'interno di un unico repository.

Come mostrato nella Figura 7.1, la struttura segue le best practice per lo stack MERN:

- **Backend:** Organizzato secondo il pattern MVC (Model-View-Controller).
- **Frontend:** Strutturato a componenti, inizializzato utilizzando Vite per garantire tempi di build rapidi e un Hot Module Replacement (HMR) efficiente.

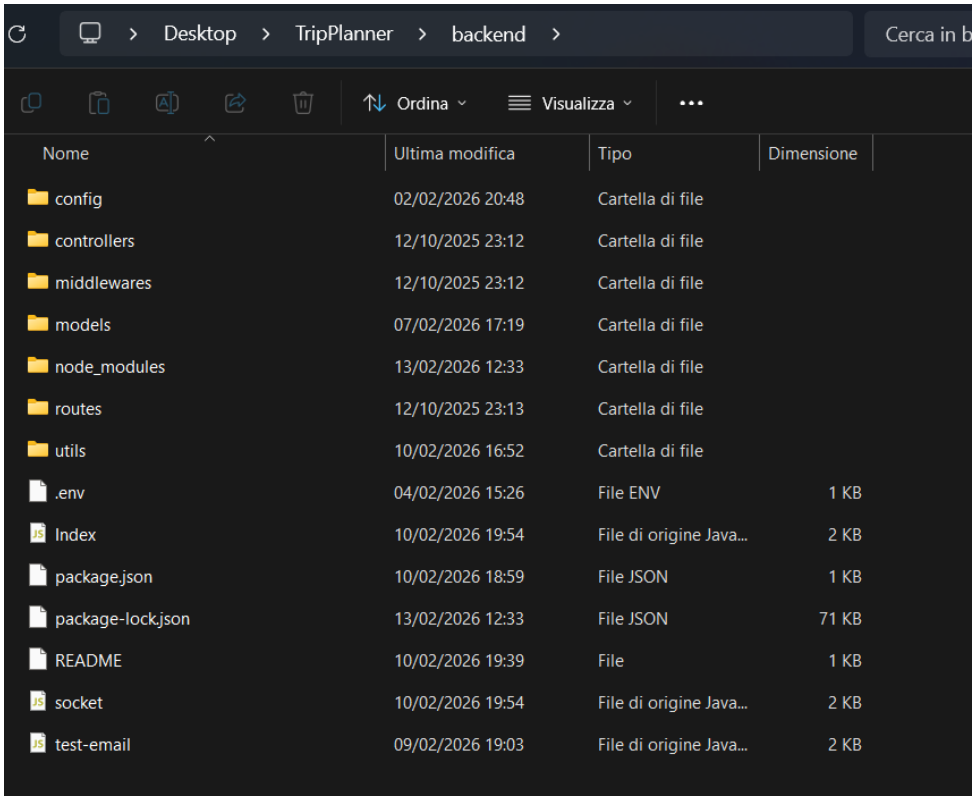

Figura 7: Struttura delle directory del progetto: dettaglio del Backend.

Analizzando nel dettaglio le directory create:

6.1.1. Struttura Backend

La cartella `server` è stata suddivisa per isolare la logica di business dalle rotte API:

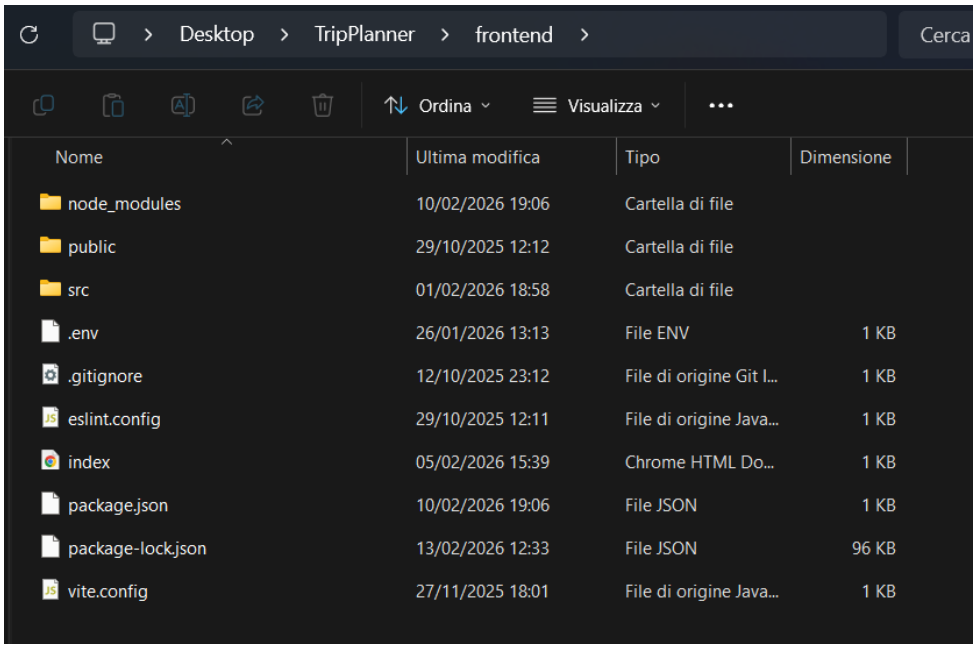

Figura 8: Struttura delle directory del progetto: dettaglio del Frontend.

- `config/`: Contiene le configurazioni del database e altre variabili d'ambiente.
- `controllers/`: Ospita la logica funzionale di ogni endpoint (es. gestione login, creazione itinerario).
- `models/`: Definisce gli Schemi Mongoose per la validazione dei dati (User, Trip).
- `routes/`: Mappa gli endpoint HTTP alle funzioni dei controller.
- `middlewares/`: Gestisce funzioni intermedie come l'autenticazione JWT.
- `utils/`: Funzioni di utilità generiche.
- `socket.js`: Un modulo dedicato separato per la gestione degli eventi in tempo reale, mantenendo il file principale `index.js` più pulito.

6.1.2. Struttura Frontend

La cartella `client` sfrutta la potenza di Vite e React:

- `public/`: Per gli asset statici.
- `src/`: Il cuore dell'applicazione, contenente tutto il codice sorgente React.
- `vite.config.js`: File di configurazione del bundler, essenziale per l'ottimizzazione delle performance.

6.2. Evoluzione dell'Interfaccia Utente (UI Evolution)

L'interfaccia di TripPlanner non è nata nella sua forma definitiva, ma è frutto di un processo di design iterativo. Partendo da wireframe a bassa fedeltà, si è passati attraverso diverse revisioni stilistiche per migliorare l'User Experience (UX) e l'estetica.

6.2.1. Fase 1: Prototipo Funzionale (Low-Fidelity)

Nelle prime fasi di sviluppo, l'obiettivo principale era testare la funzionalità delle API e il flusso dei dati, senza preoccuparsi del design. Come visibile nella Figura 7.2, la prima versione presentava un layout rudimentale, basato su HTML grezzo e CSS minimo. Questa fase è stata cruciale per verificare la corretta integrazione della mappa e il sistema di autenticazione.

Figura 9: Prima iterazione del prototipo: focus sulla funzionalità di base senza styling avanzato.

6.2.2. Fase 2: Definizione dello Stile (Mid-Fidelity)

Una volta consolidate le funzionalità base, si è passati alla definizione dell'identità visiva. È stato introdotto un logo, una palette di colori definita (toni del blu e bianco) e illustrazioni vettoriali per rendere l'accoglienza più amichevole.

La Figura 7.3 mostra questa fase intermedia: la struttura della Homepage è definita, con una "Call to Action" chiara ("Pronto per partire?"), ma il design system non è ancora completamente integrato con le funzionalità avanzate come il tema scuro.

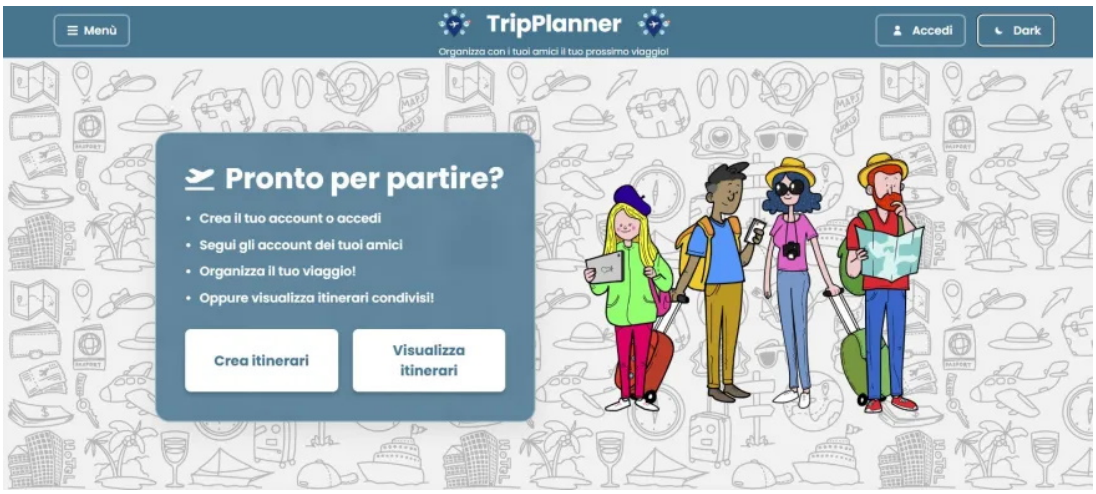

Figura 10: Seconda iterazione: introduzione del branding e miglioramento del layout grafico.

6.2.3. Fase 3: Versione Finale e Dark Mode (High-Fidelity)

L'ultima fase di sviluppo dell'interfaccia ha portato alla versione finale dell'applicazione. Le migliorie principali includono:

1. **Modernizzazione della UI:** Utilizzo di componenti più puliti, font leggibili e spaziature ottimizzate.
2. **Dark Mode:** Implementazione di un tema scuro completo (gestito tramite variabili CSS e React Context), fondamentale per ridurre l'affaticamento visivo durante la pianificazione notturna dei viaggi.
3. **Responsive Design:** La barra di navigazione e i contenuti si adattano fluidamente alle dimensioni dello schermo.

Figura 11: Versione finale dell'interfaccia in modalità Dark Mode, con design responsivo e palette colori definitiva.

7. Validazione e Verifica dei Dati

La stabilità e la sicurezza di TripPlanner dipendono da una rigorosa gestione dei dati in ingresso. In questo capitolo vengono analizzati i meccanismi di validazione implementati per garantire l'integrità delle informazioni e guidare l'utente verso un utilizzo corretto dell'applicazione.

Il sistema adotta una strategia di "Defense in Depth" (difesa in profondità), applicando controlli su due livelli distinti:

1. **Validazione Client-Side (Frontend):** Per un feedback immediato e una migliore User Experience (UX).
2. **Validazione Server-Side (Backend):** Per la sicurezza e la coerenza del database (indispensabile poiché il client non è mai affidabile).

7.1. Validazione Lato Client (Frontend)

Il primo livello di controllo avviene direttamente nel browser dell'utente tramite React. L'obiettivo è prevenire l'invio di richieste inutili al server se i dati sono palesemente errati.

7.1.1. Controllo dei Form (Input Validation)

Durante la compilazione dei moduli (Login, Registrazione, Creazione Viaggio), l'applicazione verifica in tempo reale il rispetto dei vincoli formali.

- **Campi Obbligatori:** L'attributo `required` e i controlli di stato in React impediscono la sottomissione del form se campi essenziali (come Nome Viaggio o Email) sono vuoti.
- **Formato Email:** Viene utilizzata una Regular Expression (Regex) per assicurare che l'email inserita rispetti lo standard (es. `utente@dominio.com`).
- **Coerenza delle Date:** Nel modulo di creazione itinerario, un controllo logico impedisce di selezionare una Data di Ritorno antecedente alla Data di Partenza. Se l'utente tenta di farlo, l'input viene resettato o appare un messaggio di avviso.

7.1.2. Feedback Visivo

Per migliorare l'usabilità, il sistema fornisce feedback visivi immediati:

- **Bordi Rossi/Verdi:** I campi di input cambiano colore in base alla validità del dato inserito.
- **Messaggi di Errore (Toast):** In caso di tentativo di azione non valida (es. "Password troppo corta"), viene mostrato un popup non intrusivo (Toast notification) che spiega chiaramente l'errore e come correggerlo.

7.2. Validazione Lato Server (Backend)

Poiché i controlli lato client possono essere aggirati (es. disabilitando JavaScript o usando strumenti come Postman), il backend esegue una validazione rigorosa prima di interagire con il database MongoDB.

7.2.1. Validazione tramite Mongoose Schema

La prima barriera di difesa nel backend è definita nei Models di Mongoose. Ogni schema impone regole strutturali che, se violate, generano un'eccezione che blocca il salvataggio.

Esempi di vincoli implementati:

- **Unicità:** Il campo email nello schema User ha la proprietà `unique: true`, impedendo la registrazione duplicata dello stesso indirizzo.
- **Tipizzazione Forte:** Se il sistema si aspetta una data (`Date`) e riceve una stringa di testo non convertibile, l'operazione viene respinta.
- **Enum:** Per campi con valori predefiniti (es. ruolo utente o stato del viaggio), viene verificato che il valore ricevuto appartenga alla lista consentita.

Di seguito vengono riportati i frammenti di codice significativi che definiscono le regole di integrità dei dati direttamente a livello di schema.

```
// Estratto da: models/User.js
const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true, // Il campo non può essere vuoto
    trim: true      // Rimuove spazi bianchi a inizio/fine
  },
  email: {
    type: String,
    required: true,
    unique: true,   // Impedisce la registrazione di email duplicate
    trim: true
  },
  password: {
    type: String,
    minlength: 6,   // Impone una complessità minima (Lunghezza)
    // La password è obbligatoria solo se l'utente non usa Google Login
    required: function() { return !this.googleId; }
  }
});
```

```
// Estratto da: models/Itinerario.js
const itinerarioSchema = new mongoose.Schema({
  titolo: {
    type: String,
    required: true,
    maxlength: 100 // Limita la lunghezza del testo per evitare abusi
  },
  partecipantiMax: {
    type: Number,
    min: 1,        // Validazione Logica: un viaggio deve avere almeno 1 posto
    default: 1
  },
```

```
  dataInizio: { type: Date, required: true },
  dataFine: { type: Date, required: true }
});
```

Come si evince dal codice, l'uso di proprietà come `min`, `maxlength` e `unique` delega al database il compito di rifiutare dati incoerenti, sollevando eccezioni che vengono poi gestite dal controller.

7.2.2. Middleware di Validazione Personalizzata

Per controlli più complessi che coinvolgono la logica di business, sono stati implementati controlli all'interno dei Controller o middleware dedicati.

Caso d'uso: Registrazione Utente
Quando arriva una richiesta POST `/api/auth/register`, il server esegue i seguenti controlli sequenziali:

1. Verifica che tutti i campi (nome, email, password) siano presenti nel corpo della richiesta.
2. Verifica che la password rispetti i criteri di complessità (lunghezza minima 6 caratteri).
3. Interroga il database per verificare se l'email esiste già.

Risultato: Se l'email esiste, restituisce errore `400 Bad Request` con messaggio "Utente già registrato".

7.3. Gestione degli Errori (Error Handling)

La gestione degli errori è centralizzata per garantire che il client riceva sempre risposte coerenti e comprensibili, evitando di esporre dettagli tecnici sensibili (Stack Trace) all'utente finale.

Il flusso di gestione prevede:

1. **Intercettazione:** I blocchi `try-catch` nei controller asincroni catturano le eccezioni.
2. **Classificazione:** L'errore viene classificato (es. Errore di Validazione, Errore di Autenticazione 401, Risorsa non trovata 404).
3. **Risposta:** Il server invia al client un oggetto JSON standardizzato:

```
{
  "success": false,
  "message": "La data di fine viaggio non può essere precedente all'inizio."
}
```

7.4. Esempio pratico di Flusso di Verifica

Di seguito è illustrato il flusso completo di verifica per l'inserimento di un nuovo viaggio:

1. **Utente:** Compila il form e clicca "Crea Viaggio".
2. **Frontend:** Controlla che il titolo non sia vuoto. (Se vuoto → Errore a video, stop).
3. **Frontend:** Invia i dati al Server (Token JWT incluso nell'header).
4. **Backend (Middleware Auth):** Verifica la validità del Token JWT. (Se scaduto → Errore 401, logout forzato).
5. **Backend (Controller):** Verifica che le date siano logiche e che la posizione geografica abbia coordinate valide.
6. **Database:** Mongoose tenta il salvataggio.
7. **Esito:** Se tutto è corretto, ritorna il nuovo oggetto Viaggio; altrimenti ritorna l'errore specifico che verrà mostrato all'utente.

8. Discussione e Analisi del Flusso Esecutivo

In questo capitolo viene analizzato il comportamento del sistema durante l'esecuzione reale, correlando le azioni compiute dall'utente sull'interfaccia grafica con la logica eseguita dal codice sottostante. Verranno esaminati i tre flussi principali: l'autenticazione, la creazione di un itinerario e la gestione della collaborazione in tempo reale.

8.1. Flusso di Autenticazione e Sicurezza

Il primo punto di contatto dell'utente con l'applicazione è la fase di registrazione o login. Questo passaggio è critico per garantire che solo gli utenti autorizzati possano accedere alle funzionalità di modifica.

Azione Utente: L'utente inserisce le proprie credenziali (email e password) nel form di registrazione e preme "Registrati".

Esecuzione del Codice (Backend): Quando il server riceve la richiesta POST, il modello `User` entra in azione. Prima ancora di salvare i dati nel database, viene attivato un middleware specifico per la crittografia della password.

Di seguito è riportato il commento logico al codice che gestisce questa fase critica:

```
// File: models/User.js - Middleware "pre-save"
userSchema.pre('save', async function (next) {
  // 1. Controllo preliminare: Se la password non è stata modificata
  // (es. stiamo aggiornando solo l'email), non serve ricalcolare l'hash.
  // Questo risparmia risorse CPU preziose.
  if (!this.password || !this.isModified('password')) {
    return next();
  }

  try {
    // 2. Generazione del "Salt": Una stringa casuale che rende unico l'hash,
    // proteggendo contro attacchi "Rainbow Table".
    const salt = await bcrypt.genSalt(10);

    // 3. Hashing: La password in chiaro viene sostituita con la versione criptata.
    // Nel database non verrà MAI salvata la password originale "123456".
    this.password = await bcrypt.hash(this.password, salt);

    next(); // Procede al salvataggio effettivo su MongoDB
  } catch (error) {
    next(error);
  }
});
```

Questo approccio garantisce che, anche in caso di violazione del database (Data Breach), le password degli utenti rimangano illeggibili.

Oltre alla fase di registrazione, il sistema deve garantire un accesso sicuro agli utenti registrati. La Figura 9.1 illustra nel dettaglio la sequenza di operazioni che avvengono durante il Login: dal controllo preliminare delle credenziali nel database, alla verifica

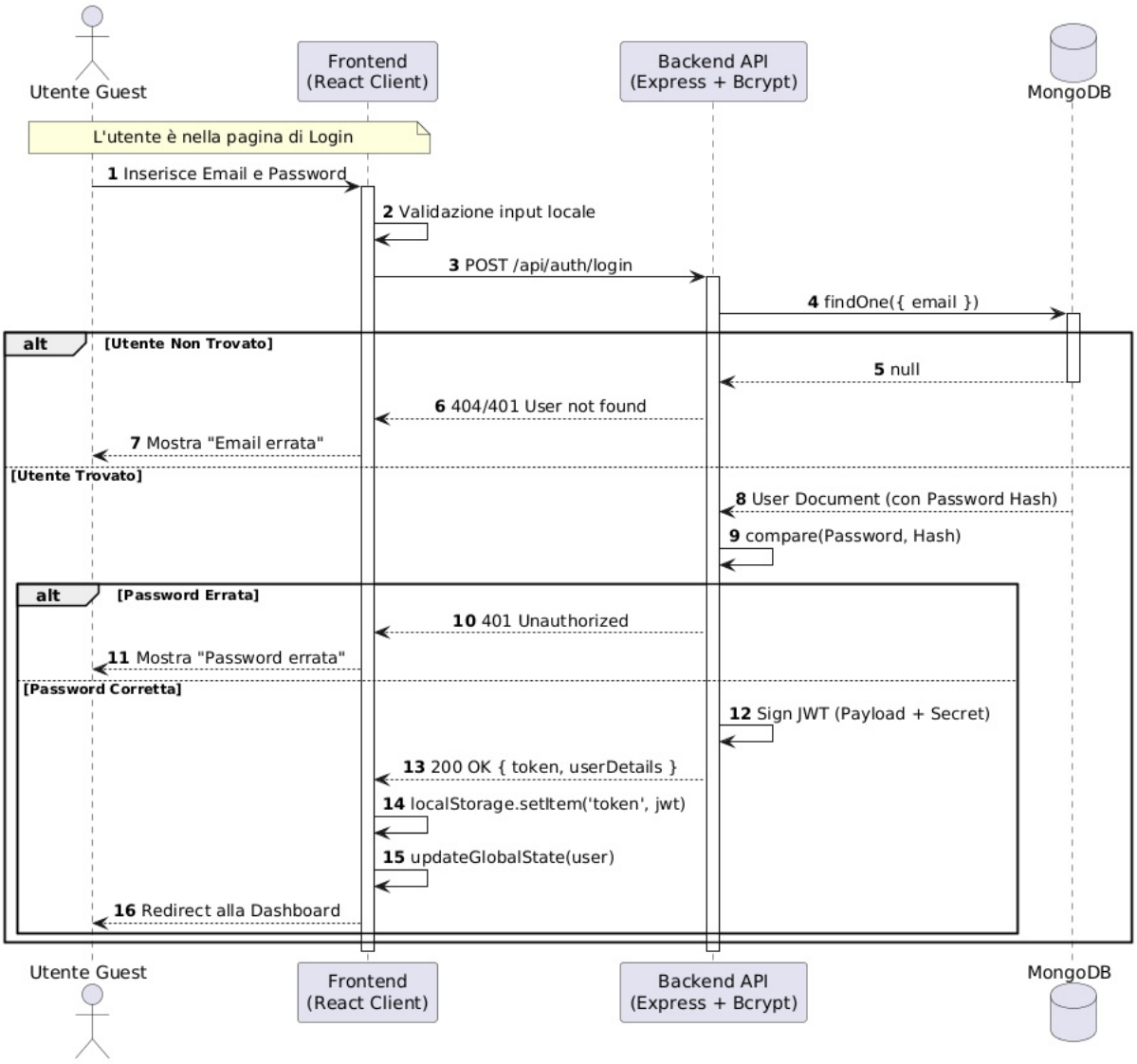

Figura 12: Diagramma di sequenza del processo di Autenticazione: verifica delle credenziali, gestione degli errori e rilascio del Token JWT.

crittografica della password tramite bcrypt, fino all'emissione del Token JWT che verrà salvato nel Local Storage del client.

8.2. Creazione e Gestione dell'Itinerario

Una volta autenticato, l'utente accede alla Dashboard. Il cuore dell'applicazione è la creazione di un nuovo viaggio.

Azione Utente: L'utente clicca su "Crea Itinerario", compila il titolo ("Viaggio a Parigi"), seleziona le date e imposta il numero massimo di partecipanti.

Esecuzione del Codice (Frontend & Backend): Lato client, React raccoglie i dati dallo stato del form. Lato server, viene istanziato un nuovo documento basato sullo schema Itinerario. È interessante notare come il codice gestisca la referenza all'autore, collegando il viaggio all'ID dell'utente loggato.

```
// Esempio Logico del Controller di creazione (Backend)
```

```
exports.createTrip = async (req, res) => {
  try {
    // 1. Estrazione dati: Recuperiamo i dati inviati dal frontend
    const { titolo, destinazione, dataInizio, dataFine, partecipantiMax } = req.body;

    // 2. Creazione Istanza: Creiamo un nuovo oggetto Itinerario.
    // Nota: req.user._id deriva dal Token JWT decodificato dal middleware di auth.
    // Questo impedisce di creare viaggi a nome di altri utenti.
    const nuovoViaggio = new Itinerario({
      titolo,
      destinazione,
      autore: req.user._id, // Collegamento relazionale (Foreign Key Logica)
      partecipantiMax,      // Impostato dall'utente (es. max 4 persone)
      stato: 'bozza'        // Valore di default definito nello schema
    });

    // 3. Persistenza: Salvataggio asincrono su MongoDB
    await nuovoViaggio.save();
    res.status(201).json(nuovoViaggio);

  } catch (error) {
    res.status(500).json({ message: "Errore nella creazione" });
  }
};
```

Il campo `partecipantiMax`, introdotto nello schema, gioca un ruolo fondamentale nel limitare le iscrizioni future: il sistema controllerà che la lunghezza dell'array `partecipantiAttuali` non superi questo numero intero.

Per comprendere appieno la complessità di questa operazione, è utile analizzare il diagramma di sequenza (Figura 9.2). Il processo include una chiamata asincrona alle API di OpenStreetMap per convertire il nome della città in coordinate geografiche (Geocoding) prima che avvenga il salvataggio effettivo su MongoDB.

8.3. Interattività della Mappa (Leaflet)

Durante la pianificazione, l'utente interagisce con la mappa per visualizzare la destinazione.

Azione Utente: L'utente digita una città o clicca sulla mappa per aggiungere una tappa.

Esecuzione del Codice (Frontend): Il componente React utilizza la libreria Leaflet. L'interazione non richiede un ricaricamento della pagina (SPA). Il codice intercetta l'evento `onClick` sulla mappa, recupera le coordinate (latitudine/longitudine) e aggiorna lo stato locale di React, che a sua volta ridisegna i "Marker" sulla mappa in tempo reale.

8.4. Collaborazione in Tempo Reale (Socket.io)

L'aspetto più innovativo discusso in questa documentazione è la gestione delle richieste di partecipazione in tempo reale.

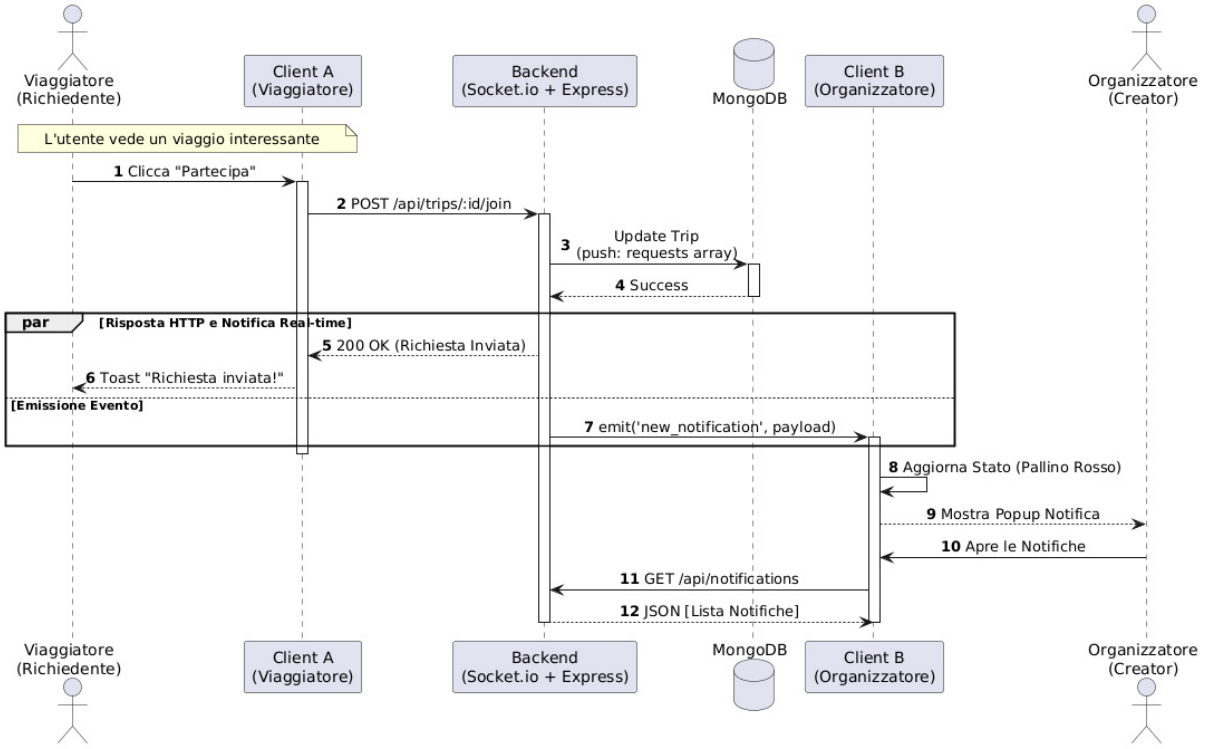

Figura 13: Diagramma di sequenza per la creazione di un nuovo viaggio: interazione tra Client, API di OpenStreetMap e Backend.

Scenario: L'Utente A (Organizzatore) è sulla pagina del viaggio. L'Utente B (Amico) clicca su "Chiedi di partecipare".

Flusso Esecutivo:

1. **Frontend (Utente B):** Emette un evento Socket (es. `join_request`) e contemporaneamente invia una chiamata API per aggiornare l'array `richiestePendenti` nel database.
2. **Backend:** Riceve la notifica e la propaga.
3. **Frontend (Utente A):** Riceve l'evento e mostra un badge di notifica senza dover aggiornare la pagina.

```
// Esempio Logico gestione Socket.io (Backend)
io.on('connection', (socket) => {
  // L'utente entra nella "stanza" specifica di quel viaggio
  socket.on('join_trip_room', (tripId) => {
    socket.join(tripId);
    console.log(`Utente connesso alla stanza del viaggio: ${tripId}`);
  });

  // Quando arriva una nuova richiesta di partecipazione
  socket.on('send_request', (data) => {
    // data contiene: { tripId, userRequestingName }
    // Invia la notifica SOLO agli utenti dentro quella stanza (il proprietario)
    socket.to(data.tripId).emit('receive_notification', {
```

```
      message: `${data.userRequestingName} vuole unirsi al viaggio!`,
      type: 'info'
    });
  });
});
```

Questo meccanismo dimostra l'efficienza dello stack scelto: MongoDB gestisce la persistenza a lungo termine (salvando la richiesta nell'array), mentre Socket.io gestisce l'esperienza utente immediata (feedback visivo istantaneo).

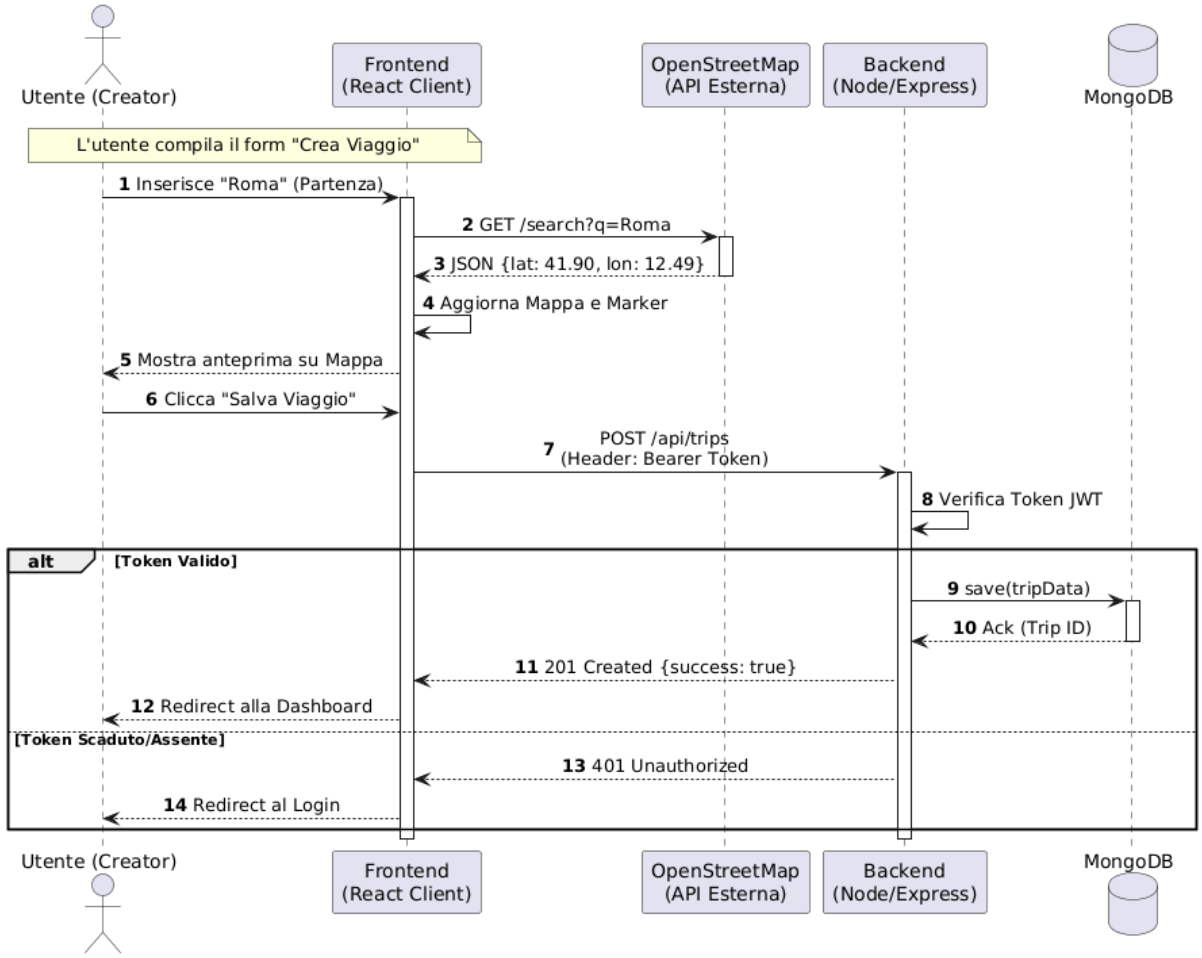

Figura 14: Diagramma di sequenza del flusso di richiesta partecipazione: sincronizzazione tra richiesta HTTP persistente e notifica volatile via Socket.io.

9. Documentazione API (Backend)

L'interfaccia di programmazione (API) del server è stata progettata seguendo i principi RESTful. Tutte le risposte sono in formato JSON. Le rotte sono prefissate da `/api` e suddivise per domini funzionali.

9.1. Autenticazione (`/api/auth`)

Gestisce il ciclo di vita dell'identità utente.

Metodo	**Endpoint**	**Auth**	**Descrizione**
`POST`	`/register`	No	Registra un nuovo utente e invia email di verifica.
`POST`	`/verify-email`	No	Verifica il token inviato via email.
`POST`	`/login`	No	Autentica l'utente e restituisce Access Token e Refresh Token.
`GET`	`/refresh`	No	Genera un nuovo Access Token usando il cookie jwt.
`GET`	`/logout`	No	Invalida il Refresh Token e cancella i cookie.
`GET`	`/me`	Si	Restituisce i dati dell'utente attualmente loggato.
`GET`	`/google`	No	Avvia il flusso di autenticazione OAuth con Google.

Tabella 1: Endpoint per la gestione dell'autenticazione.

9.2. Gestione Itinerari (`/api/itinerari`)

Queste rotte gestiscono le operazioni CRUD sui viaggi e le funzionalità di ricerca.

9.3. Social & Notifiche (`/api/social`)

Gestisce le interazioni tra utenti e la collaborazione in tempo reale.

Metodo	Endpoint	Auth	Descrizione
POST	/crea	Si	Crea un nuovo itinerario nel database.
GET	/miei	Si	Restituisce la lista dei viaggi creati dall'utente corrente.
GET	/visualizza	No	Restituisce la lista pubblica degli itinerari (per la Home).
GET	/visualizza/:id	No	Restituisce i dettagli completi di uno specifico viaggio.
PUT	/modifica/:id	Si	Aggiorna i dati di un viaggio esistente (solo Creator).
DELETE	/cancella/:id	Si	Elimina definitivamente un itinerario (solo Creator).
GET	/ricerca	No	Cerca viaggi pubblici filtrando per destinazione o titolo.

Tabella 2: Endpoint per la gestione degli itinerari.

Metodo	Endpoint	Auth	Descrizione
POST	/partecipa/:id	Si	Invia una richiesta di partecipazione a un viaggio.
POST	/accetta	Si	Il Creator accetta una richiesta pendente.
POST	/rifiuta	Si	Il Creator rifiuta una richiesta pendente.
POST	/abbandona/:id	Si	Un partecipante abbandona volontariamente il viaggio.
POST	/rimuovi-partecipante	Si	Il Creator rimuove forzatamente un utente (Kick).
POST	/accetta-invito	Si	Un utente accetta l'invito ricevuto via email.
GET	/notifiche	Si	Recupera la lista delle notifiche dell'utente.
PUT	/notifiche/:id/leggi	Si	Segna una specifica notifica come letta.

Tabella 3: Endpoint per le funzionalità social e notifiche.

10. Descrizione Componenti React (Frontend)

L'architettura frontend separa le Pagine (*Container*) dai Componenti UI (*Presentational*). Di seguito vengono descritti i componenti riutilizzabili suddivisi per area funzionale.

10.1. Componenti Strutturali e di Navigazione

Questi componenti definiscono il layout e la navigazione trasversale dell'applicazione.

- `Header.jsx`: Barra superiore responsive. Gestisce il toggle del tema (Light/Dark), il menu hamburger per mobile e mostra il badge di notifica.
- `Sidebar.jsx`: Menu laterale a scomparsa.
  - Decodifica autonomamente il token JWT (tramite `atob`) per mostrare nome ed email dell'utente senza dover fare una chiamata API dedicata.
  - Gestisce il rendering condizionale delle voci di menu (Utente loggato vs Ospite).
  - Include la logica di Logout che pulisce il LocalStorage e disconnette la socket.
- `Footer.jsx`: Footer dinamico che si nasconde nelle pagine di login/register e adatta i link in base allo stato di autenticazione.
- `SharedMap.jsx`: Wrapper avanzato per React-Leaflet.
  - Risolve il noto bug delle icone di default di Leaflet importandole manualmente.
  - Utilizza un hook interno `MapUpdater` per adattare automaticamente lo zoom (*bounds*) in modo da includere tutti i marker visibili.
  - Visualizza sia i marker delle tappe che la polilinea del percorso (`routePath`).

10.2. Componenti di Gestione Viaggi (Dashboard & Home)

Componenti utilizzati per visualizzare le liste di viaggi e le card.

- `TripCard.jsx`: Componente card versatile.
  - Accetta una prop `actionButtons` che permette di iniettare bottoni diversi a seconda del contesto (es. "Modifica/Elimina" nella Dashboard, "Vedi" nella Home).
  - Determina dinamicamente l'icona del mezzo di trasporto (aereo, treno, auto) e gestisce il fallback per l'avatar dell'autore.
- `TripList.jsx`: Utilizzato nella Homepage. Esegue il fetch dei viaggi pubblici, li filtra e mostra solo i primi risultati (*slice*) come anteprima per i visitatori.
- `DashboardGrid.jsx` & `ExploreGrid.jsx`: Wrapper che organizzano le `TripCard` in una griglia responsive CSS Grid, gestendo gli stati di caricamento e "nessun risultato".

10.3. Componenti per la Creazione Viaggio (Form Wizard)

Utilizzati nella pagina `CreateTrip.jsx` per spezzare la complessità del form.

- `TripForm.jsx`: Gestisce l'input dati complesso.
  - Include la logica per aggiungere/rimuovere dinamicamente le tappe intermedie e gli inviti agli amici (array di stringhe).
  - Fornisce un selettore visivo per il mezzo di trasporto.
- `TripPreview.jsx`: Pannello laterale che offre un feedback visivo immediato. Mostra la mappa aggiornata in tempo reale e il calcolo stimato della durata del viaggio mentre l'utente compila il form.

10.4. Componenti Dettaglio Viaggio

Utilizzati esclusivamente nella pagina `TripDetails.jsx` per mostrare le informazioni specifiche.

- `TripHeader.jsx`: Visualizza il titolo macroscopico e i badge di stato (Mezzo di trasporto e Disponibilità posti colorata in verde/rosso).
- `TripInfo.jsx`: Pannello informativo.
  - Mostra date, tappe e lista partecipanti.
  - Contiene la logica condizionale per il "Bottone Principale": cambia testo e funzione se l'utente è l'autore, un partecipante, un invitato o un estraneo.

10.5. Componenti di Autenticazione e Feedback

- `LoginForm.jsx` & `RegisterForm.jsx`: Form puri che gestiscono la validazione degli input e includono il pulsante "Accedi con Google".
- `LoginHeader.jsx` & `RegisterHeader.jsx`: Componenti presentazionali per le intestazioni delle pagine di auth.
- `VerifyMessage.jsx`: Componente di stato per la verifica email. Mostra icone grandi animate (spinner, spunta verde, errore rosso) per dare un feedback chiaro all'utente durante il processo di attivazione account.

11. Conclusioni e Sviluppi Futuri

Il progetto TripPlanner ha permesso di realizzare una piattaforma funzionale per la gestione collaborativa dei viaggi, dimostrando come le tecnologie web moderne possano risolvere problemi organizzativi complessi. In questa sezione finale vengono analizzati i risultati ottenuti, le limitazioni attuali e le prospettive di evoluzione del software.

11.1. Raggiungimento degli Obiettivi

Confrontando il prodotto finale con i requisiti iniziali, il bilancio è positivo. La piattaforma soddisfa tutti i requisiti funzionali (Core):

- **Gestione Completa del Viaggio:** È possibile creare, modificare e cancellare itinerari con tappe multiple.
- **Interattività Geografica:** L'integrazione con Leaflet e OSRM fornisce un contesto spaziale immediato e calcoli di routing automatici.
- **Collaborazione Real-Time:** L'uso di Socket.io ha trasformato l'esperienza utente, permettendo notifiche istantanee per le richieste di partecipazione, eliminando la necessità di refresh manuali.
- **Sicurezza:** L'implementazione di JWT con Refresh Token e l'hashing delle password garantisce standard di sicurezza adeguati.

11.2. Limitazioni Attuali

Nonostante il successo del prototipo, esistono aree di miglioramento identificate durante la fase di testing:

- **Scalabilità del Database:** Attualmente MongoDB opera su una singola istanza. In uno scenario di produzione con migliaia di utenti, sarebbe necessario implementare strategie di *Sharding* o *Replication*.
- **Accessibilità:** Sebbene l'interfaccia sia intuitiva, non è stata ancora effettuata una verifica completa della conformità agli standard WCAG (Web Content Accessibility Guidelines) per utenti con disabilità visive.
- **Chat Interna:** La comunicazione testuale tra i partecipanti è ancora demandata a strumenti esterni (WhatsApp), frammentando parzialmente l'esperienza.

11.3. Sviluppi Futuri

Per trasformare TripPlanner da progetto accademico a prodotto commerciale (MVP), la roadmap di sviluppo prevede le seguenti direzioni prioritarie:

11.3.1. Deployment e Distribuzione

Il primo obiettivo per la messa in produzione riguarda la pubblicazione dell'applicazione su un'infrastruttura cloud.

- **Stato attuale:** Sono stati effettuati test preliminari di deployment (utilizzando servizi come Heroku/Vercel per il frontend e istanze AWS/DigitalOcean per il backend); tuttavia, a causa di complessità nella configurazione delle variabili d'ambiente e della persistenza del database in ambiente di produzione, il processo non è stato ancora finalizzato con successo.
- **Obiettivo:** Configurazione di una pipeline di CI/CD (Continuous Integration/Continuous Deployment) per automatizzare il rilascio e garantire l'accessibilità pubblica della piattaforma.

11.3.2. Gamification e Trust System

Per incentivare l'uso della piattaforma, si prevede l'introduzione di elementi di gioco:

- **Livelli Utente:** Assegnazione di badge (es. "Viaggiatore Esperto", "Organizzatore Top") basati sul numero di viaggi completati.
- **Recensioni:** Possibilità di lasciare feedback sui compagni di viaggio post-esperienza. Un profilo con recensioni positive aumenterà la fiducia (*Trust*) all'interno della community.

11.3.3. Integrazioni Funzionali Avanzate

- **Chat di Gruppo Integrata:** Sfruttando l'infrastruttura Socket.io già esistente, verranno create "stanze" di chat persistenti per ogni viaggio, centralizzando l'organizzazione logistica.
- **API di Terze Parti (Meteo e Booking):** Arricchimento delle card di viaggio con previsioni meteo in tempo reale (tramite OpenWeatherMap) per le date selezionate e suggerimenti automatici per voli e alloggi (tramite API come Skyscanner o Amadeus), offrendo un'esperienza "All-in-One".

11.3.4. Porting Mobile (React Native)

Sfruttando la base di codice React esistente, il passo successivo naturale è lo sviluppo di un'app mobile nativa. Questo permetterebbe di accedere a funzionalità hardware del dispositivo come:

- **GPS:** Per la geolocalizzazione in background durante il viaggio.
- **Fotocamera:** Per caricare foto dei luoghi direttamente nell'itinerario.
- **Notifiche Push:** Per avvisi di sistema anche ad app chiusa.

Conclusione personale del gruppo

In conclusione, lo sviluppo di TripPlanner ha rappresentato un'importante opportunità di crescita professionale per tutto il team. Ci ha permesso di approfondire tecnologie all'avanguardia nello sviluppo web e di comprendere le complessità legate alla progettazione di sistemi distribuiti e collaborativi. La sfida di coordinare frontend, backend e database in un'architettura coerente ha consolidato le nostre competenze di problem-solving, ponendo solide basi per il nostro futuro professionale.

12. Bibliografia e Sitografia

12.1. Documentazione Ufficiale (Primary Sources)

Queste sono le fonti essenziali perché descrivono come funzionano le tecnologie che sono state utilizzate.

- **MongoDB Inc.**, *MongoDB Manual.*
  Disponibile su: https://www.mongodb.com/docs/manual/
- **Express.js Community**, *Express - Fast, unopinionated, minimalist web framework for Node.js.*
  Disponibile su: https://expressjs.com/
- **Meta Open Source**, *React - The library for web and native user interfaces.*
  Disponibile su: https://react.dev/
- **OpenJS Foundation**, *Node.js Documentation.*
  Disponibile su: https://nodejs.org/en/docs/
- **Automattic**, *Mongoose ODM v8.0.0 Documentation.*
  Disponibile su: https://mongoosejs.com/docs/
- **Socket.IO**, *Socket.IO Documentation: Bidirectional and low-latency communication.*
  Disponibile su: https://socket.io/docs/v4/

12.2. Librerie e Componenti Specifici

In questa sezione sono riportate le librerie "speciali" utilizzate per mappe, sicurezza e gestione mail.

- **Leaflet**, *An open-source JavaScript library for mobile-friendly interactive maps.*
  Disponibile su: https://leafletjs.com/
- **OpenStreetMap**, *Nominatim API (Geocoding).*
  Disponibile su: https://nominatim.org/
- **Project OSRM**, *Open Source Routing Machine (Routing API).*
  Disponibile su: http://project-osrm.org/
- **Passport.js**, *Authentication middleware for Node.js (Google OAuth Strategy).*
  Disponibile su: https://www.passportjs.org/
- **Auth0**, *JSON Web Token (JWT) Introduction.*
  Disponibile su: https://jwt.io/introduction
- **Nodemailer**, *Send e-mails from Node.js.*
  Disponibile su: https://nodemailer.com/

12.3. Risorse Teoriche e Guide di Riferimento

Risorse utilizzate per giustificare le scelte architetturali (REST, SPA, CSS).

- **Mozilla Developer Network (MDN)**, *Web Docs: HTML, CSS & JavaScript Reference.*
  Disponibile su: https://developer.mozilla.org/
- **Fielding, R. T. (2000)**. *Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, Irvine. (Testo di riferimento per il modello REST).
- **Vite**, *Next Generation Frontend Tooling.*
  Disponibile su: https://vitejs.dev/

12.4. Strumenti di Sviluppo (Opzionale)

- **Postman**, *API Platform for building and using APIs.*
- **Visual Studio Code**, *Code Editing. Redefined.*